МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Кафедра вычислительных технологий**

**ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №8**

**по дисциплине Методы поисковой оптимизации**

Работу выполнила\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ромашкина А.А.

Факультет Компьютерных технологий и прикладной математики

Направление подготовки 02.03.02 Фундаментальная информатика и

информационные технологии курс 4

Краснодар

2022

СОДЕРЖАНИЕ

[Задание 3](#_Toc121830576)

[Алгоритм 3](#_Toc121830577)

[Результат работы программы 4](#_Toc121830578)

[Листинг 5](#_Toc121830579)

[Файл lab8\_bacteria.py 5](#_Toc121830580)

[Файл lab8\_immune\_network.py 7](#_Toc121830581)

Тема работы: Гибридный алгоритм.

# Задание

Необходимо разработать гибридный алгоритм.

# Алгоритм

1. *Этап иммунной сети.* Для глобального поиска используется алгоритм иммунной сети. После заданного числа итераций из итоговой популяции антител выбирается некоторое количество лучших, которые используются на следующем этапе.
2. *Этап бактериальной оптимизации.*
   1. *Инициализация популяции*: определение векторов начальных положений бактерий , величин шагов хемотаксиса , а также направляющего вектора .
   2. *Хемотаксис*. Вычисляется следующее положение бактерии по формуле:

При плавании бактерии на следующей итерации вектор остается неизменным. При кувырке бактерии вектор представляет собой случайный вектор, значения компонент которого находятся в интервале [-1; 1]. Плавание каждой из бактерий продолжается до тех пор, пока происходит уменьшение значений фитнес-функции.

Величина шага хемотаксиса меняется на случайное значение из интервала [0; 0,5] на каждой итерации алгоритма.

* 1. Считаем количество подряд идущих итераций, на которых значение фитнес-функции у бактерии на лучшей позиции оказывается больше, чем у лучшей бактерии на предыдущей итерации. Если количество таких итераций превышает разрешенное, выполняется процедура репродукции.
  2. *Репродукция*. Механизм репродукции ускоряет сходимость алгоритма. Назовем текущим состоянием здоровья бактерии сумму значений фитнес-функции во всех точках ее траектории от первой до текущей итерации.

Вычислим значения всех величин , отсортируем бактерии в порядке возрастания состояний их здоровья и представим результат сортировки в виде линейного списка. Механизм репродукции состоит в том, что вторая половина агентов (наиболее слабых) исключается из указанного списка, а каждый из агентов первой половины списка расщепляется на два одинаковых агента с одинаковыми координатами, равными координатам расщепленного агента.

# Результат работы программы

Вся работа программы представлена на рисунках 1-2 ниже.

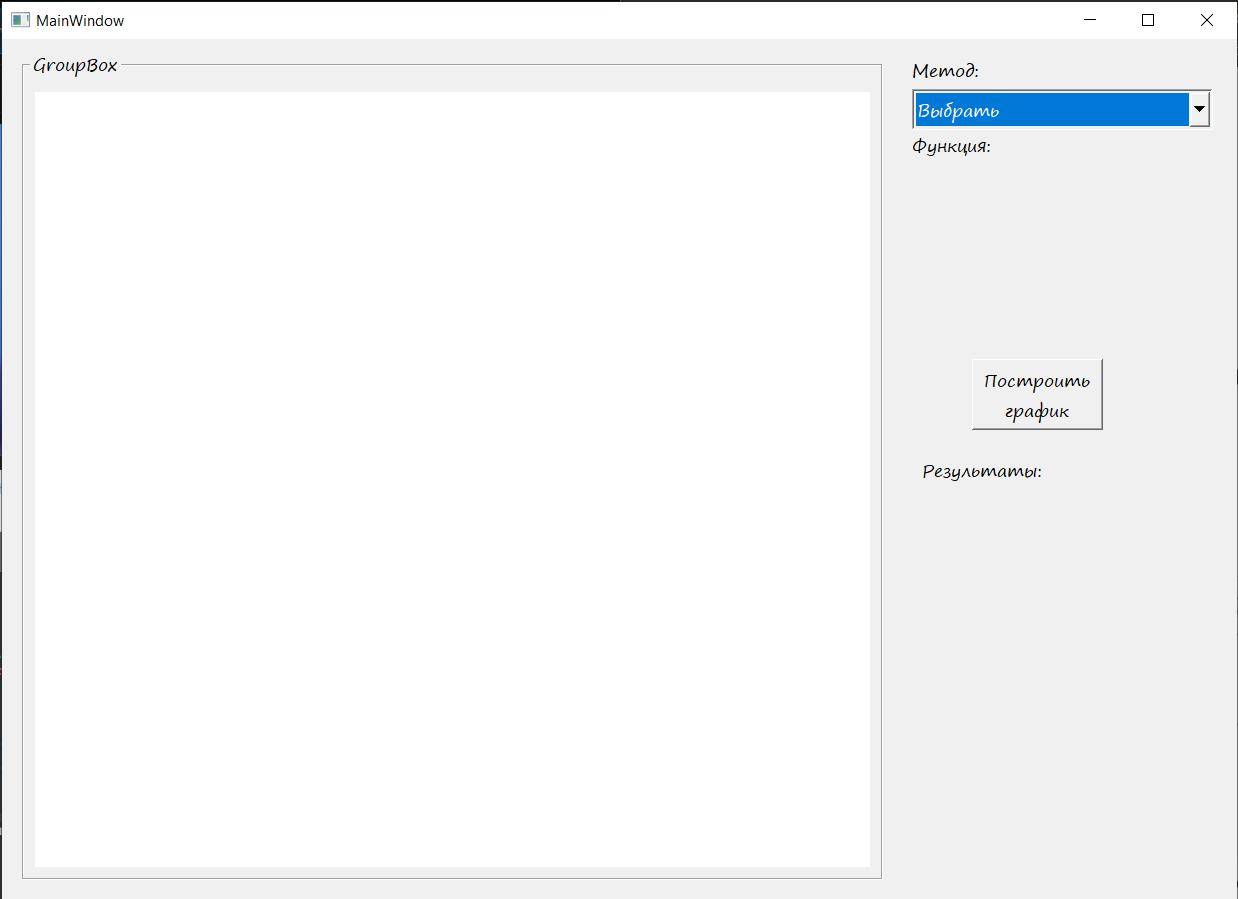


Рисунок 1 – Начальный вид программы при запуске

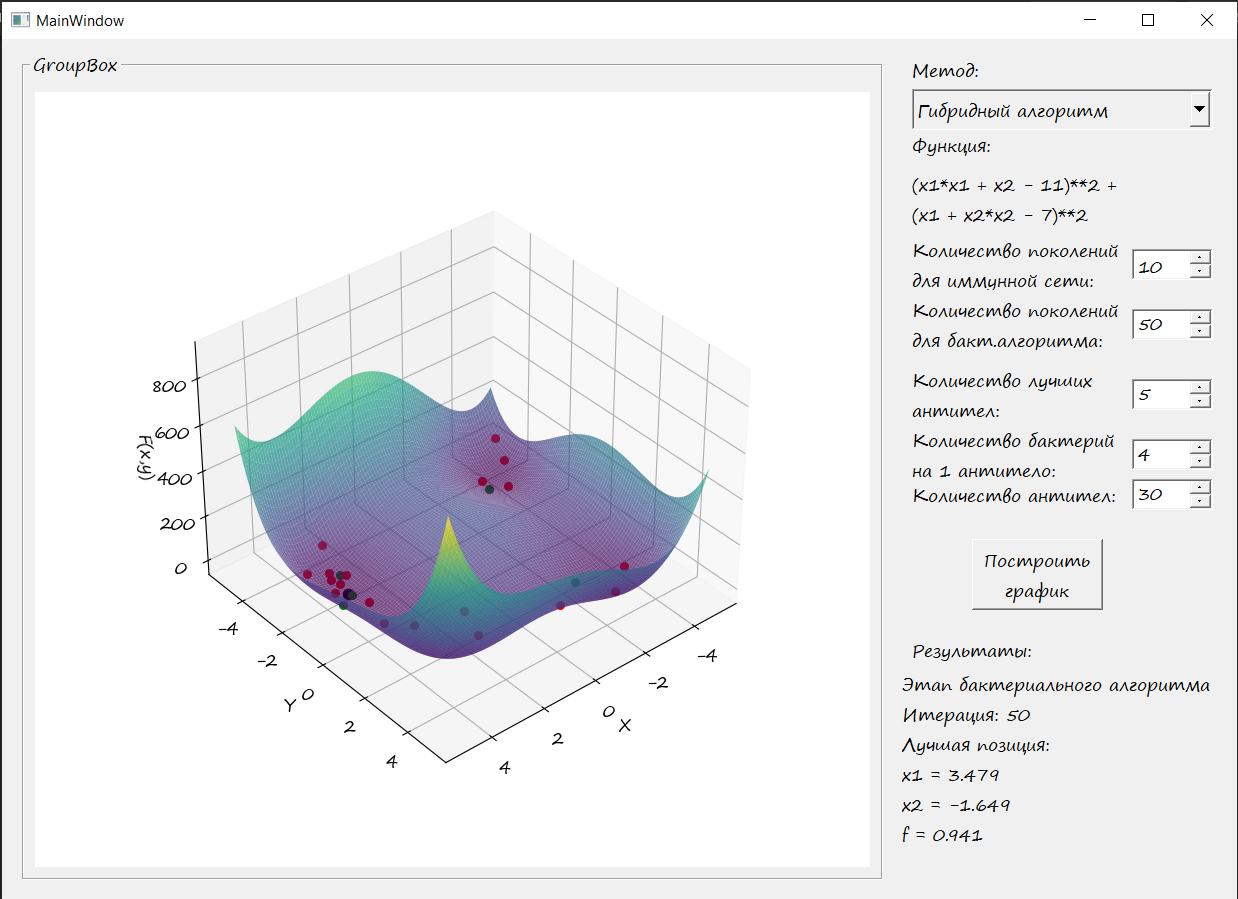


Рисунок 2 – Вид программы при выборе метода

На рисунках 3, 4 показаны графики работы алгоритма бактериальной оптимизации для 50 итераций и 70 бактерий. На рисунке 3 показана зависимость лучших значений от времени работы программы, на рисунке 4 – от номера итераций.

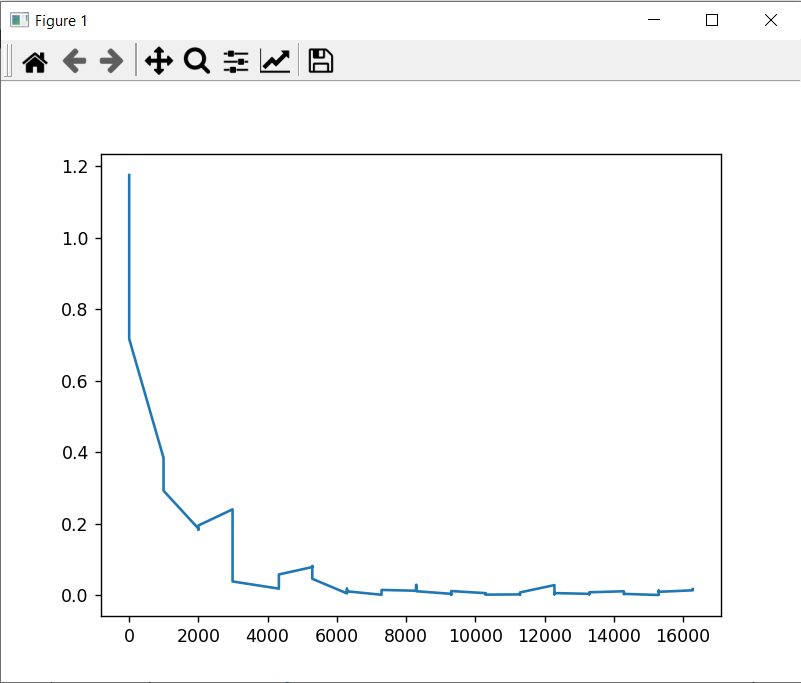
****

Рисунок 3 – График работы алгоритма (зависимость от времени)

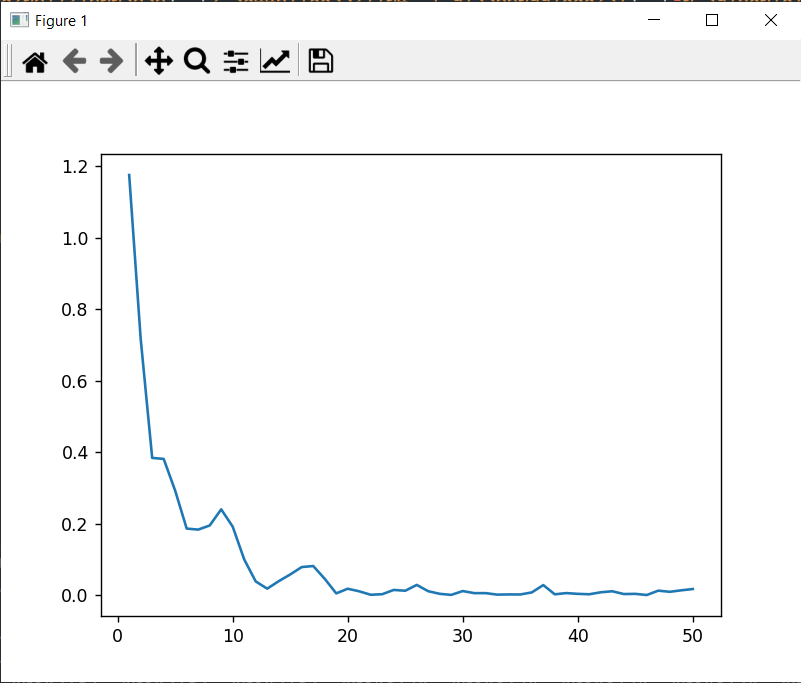


Рисунок 4 – График работы алгоритма (зависимость от итерации)

На рисунках 5, 6 показаны графики работы алгоритма иммунной сети для 50 итераций, 70 антител и 10 антигенов. На рисунке 5 показана зависимость лучших значений от времени работы программы, на рисунке 6 – от номера итераций.

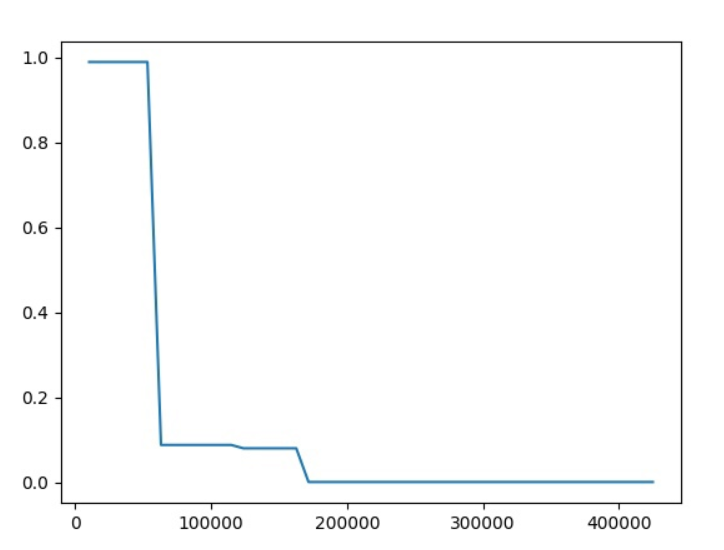


Рисунок 5 – График работы алгоритма (зависимость от времени)

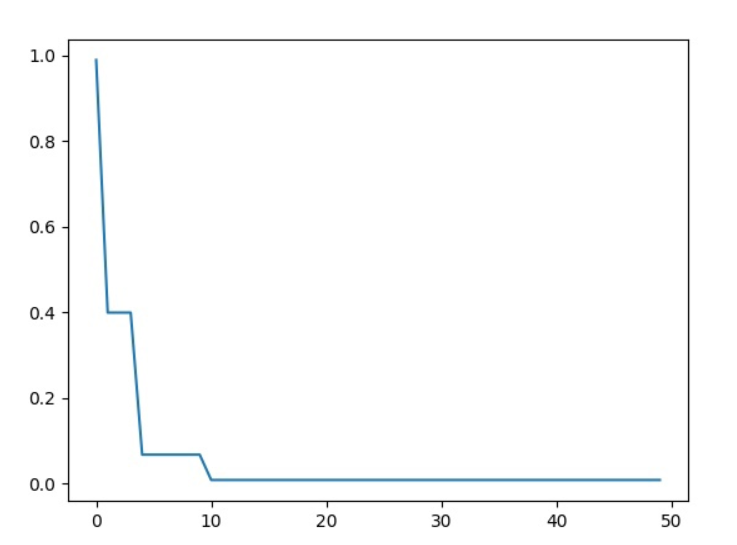


Рисунок 6 – График работы алгоритма (зависимость от итерации)

По рисункам видно, что алгоритм бактериальной оптимизации успевает проверить больше разных точек, что помогает ему в конце находить более оптимальное решение, поэтому мы используем его для локального поиска.

На рисунках 7, 8 показаны графики работы гибридного алгоритма для 20 итераций первого этапа, 30 итераций второго этапа, 60 бактерий, 70 антител и 10 антигенов. На рисунке 7 показана зависимость лучших значений от времени работы программы, на рисунке 8 – от номера итераций.

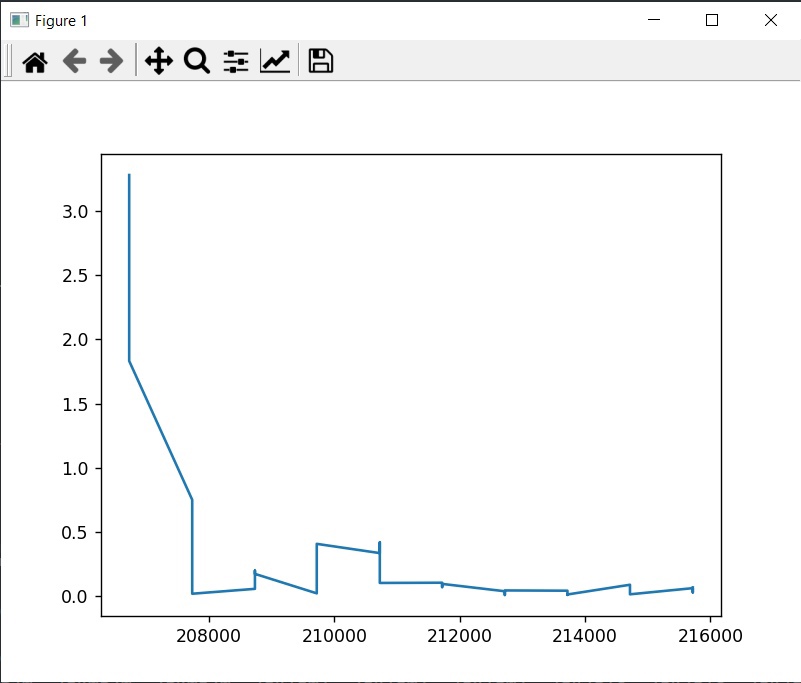


Рисунок 7 – График работы алгоритма (зависимость от времени)

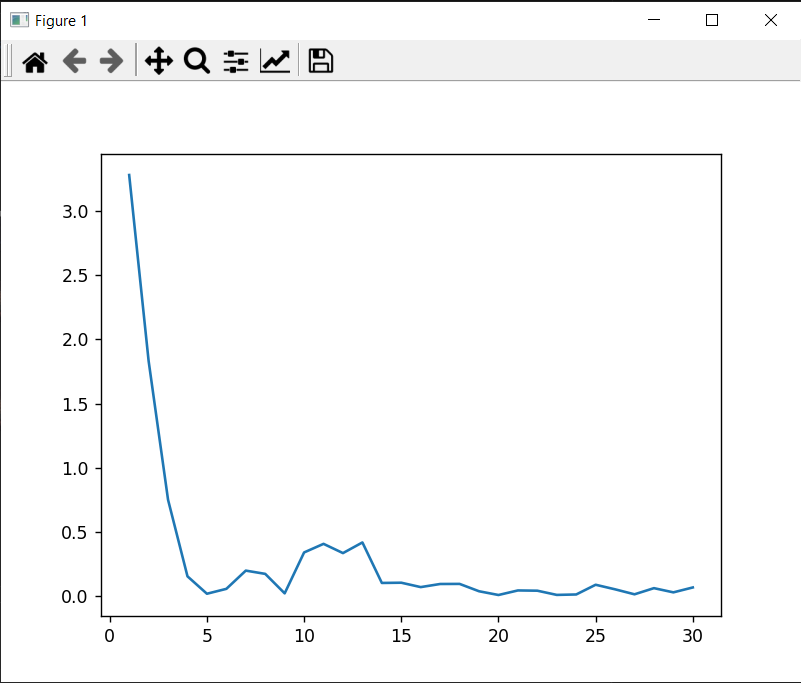


Рисунок 8 – График работы алгоритма (зависимость от итерации)

По графикам видно, что гибридный алгоритм находит достаточно хорошее решение за небольшое число итераций бактериального алгоритма. По сравнению с отдельными алгоритмами иммунной сети и бактериальной оптимизации гибридный алгоритм работает немного быстрее.

# Листинг

# Файл lab8\_bacteria.py

import numpy as np

import random

from lab7\_bacteria import Bacterium

class BacteriaAlgoritm2:

def \_\_init\_\_(self,bacterium,best\_imm,count\_for\_one\_imm,count\_error,radius):

self.count\_error = count\_error

self.error = 0

self.bacterium = bacterium

self.population = self.make\_start\_population(best\_imm,count\_for\_one\_imm,radius)

self.best\_bacterium = None

self.old\_best\_bacterium = None

self.find\_best\_bacterium()

def make\_start\_population(self,best\_imm,count\_for\_one\_imm,radius):

population = []

for imm in best\_imm:

for j in range(count\_for\_one\_imm):

population.append(self.bacterium(imm, radius))

return population

def find\_best\_bacterium(self):

self.old\_best\_bacterium = self.best\_bacterium

self.best\_bacterium = min(self.population, key=lambda x: x.fitness)

if self.old\_best\_bacterium != None:

if self.best\_bacterium.fitness > self.old\_best\_bacterium.fitness:

self.error += 1

else: self.error = 0

def get\_positions(self):

return [i.position + [i.fitness] for i in self.population]

def norm(self,vector):

res = 0

for i in vector:

res += i\*\*2

return np.sqrt(res)

def chemotaxis(self):

for i in range(len(self.population)):

#кувырок:

if self.population[i].old\_fitness < self.population[i].fitness:

self.population[i].velocity = self.population[i].make\_new\_velocity()

self.population[i].position[0] += self.population[i].step \* (self.population[i].velocity[0]/self.norm(self.population[i].velocity))

self.population[i].position[1] += self.population[i].step \* (self.population[i].velocity[1]/self.norm(self.population[i].velocity))

self.population[i].checkposition()

self.population[i].calc\_fitness()

self.population[i].step = self.population[i].make\_new\_step()

def reproduction(self):

self.population.sort(key=lambda x: x.health, reverse=False)

self.population = self.population[:len(self.population)//2]

self.population += self.population.copy()

for i in range(len(self.population)//2,len(self.population)):

self.population[i].velocity = self.population[i].make\_new\_velocity()

self.population[i].calc\_fitness()

self.population[i].step /= 2

def next\_iteration(self):

self.chemotaxis()

self.find\_best\_bacterium()

if self.error > self.count\_error:

self.reproduction()

self.find\_best\_bacterium()

class HimmelblauBacterium(Bacterium):

def \_\_init\_\_(self, imm\_cell ,radius):

Bacterium.\_\_init\_\_(self)

self.my\_imm\_cell = imm\_cell

self.radius = radius

self.max = 5

self.min = -5

self.position = self.make\_start\_position()

self.calc\_fitness()

def calc\_fitness(self):

self.old\_fitness = self.fitness

self.fitness = HimmelblauBacterium.fun(self.position[0],self.position[1])

self.health += self.fitness

def make\_start\_position(self):

return [random.uniform(self.my\_imm\_cell[0]-self.radius,self.my\_imm\_cell[0]+self.radius), random.uniform(self.my\_imm\_cell[1]-self.radius,self.my\_imm\_cell[1]+self.radius)]

def make\_new\_step(self):

return random.uniform(0,0.5)

@staticmethod

def fun(x1,x2):

return (x1\*x1 + x2 - 11)\*\*2 + (x1 + x2\*x2 - 7)\*\*2

@staticmethod

def graph():

x1 = np.linspace(-5, 5, 100)

x2 = np.linspace(-5, 5, 100)

x1,x2 = np.meshgrid(x1, x2)

f = np.array([HimmelblauBacterium.fun(x1[i],x2[i]) for i in range(len(x1))])

return x1,x2,f

# Файл lab8\_immune\_network.py

import math

import random

class Antibody:

def \_\_init\_\_(self, x\_value, y\_value):

self.x = x\_value

self.y = y\_value

class Antigen:

def \_\_init\_\_(self, x\_value, y\_value):

self.x = x\_value

self.y = y\_value

class Population:

def \_\_init\_\_(self):

self.individuals = None

def create\_population\_of\_antibodies(self, population\_size, a, b):

self.individuals = [Antibody(random.uniform(a, b), random.uniform(a, b)) for \_ in range(population\_size)]

def create\_population\_of\_antigens(self, population\_size, a, b):

self.individuals = [Antigen(random.uniform(a, b), random.uniform(a, b)) for \_ in range(population\_size)]

def reduce\_population(self, min\_affinity, f):

flag = True

population\_size = len(self.individuals)

while flag:

flag = False

for i in range(population\_size):

for j in range(i + 1, population\_size):

if self.individuals[i] is not None and self.individuals[j] is not None:

if affinity(self.individuals[i], self.individuals[j]) < min\_affinity:

flag = True

if f(self.individuals[i].x, self.individuals[i].y) < f(self.individuals[j].x, self.individuals[j].y):

self.individuals[j] = None

else:

self.individuals[i] = None

self.individuals = list(filter(lambda a: a is not None, self.individuals))

def to\_list(self):

list\_individuals\_of\_population = []

for i in self.individuals:

list\_individuals\_of\_population.append([i.x, i.y])

return list\_individuals\_of\_population

# nb - количество антител, которые будут подвергнуты мутации

# nc - число клонов клонируемого антитела

# nd - число оставляемых клонов

# bb - пороговый коэффициент гибели

# br - коэффициент клонального сжатия

# iterations - количество итераций алгоритма

class ImmuneNetworkAlgorithm:

def \_\_init\_\_(self, f, nb, nd, nc, bb, br):

self.f = f

self.nb = nb

self.nd = nd

self.nc = nc

self.bb = bb

self.br = br

def create\_memory\_cells(self, population\_of\_antibodies, population\_of\_antigens):

for antigen in population\_of\_antigens.individuals:

best\_antibodies = self.get\_best\_antibodies(population\_of\_antibodies, antigen, self.nb)

population\_memory\_cells = self.clone\_and\_mutate(best\_antibodies, self.nc, self.nd, antigen, self.bb, self.br)

for memory\_cells in population\_memory\_cells.individuals:

population\_of\_antibodies.individuals.append(memory\_cells)

population\_of\_antibodies.reduce\_population(self.br, self.f)

return population\_of\_antibodies

@staticmethod

def get\_best\_antibodies(population\_of\_antibodies, antigen, nb):

population\_of\_best\_antibodies = Population()

population\_of\_best\_antibodies.individuals = []

population\_of\_antibodies.individuals.sort(key=lambda x: affinity(x, antigen), reverse=False)

for i in population\_of\_antibodies.individuals[:nb]:

population\_of\_best\_antibodies.individuals.append(i)

return population\_of\_best\_antibodies

def clone\_and\_mutate(self, population\_of\_antibodies, nc, nd, antigen, bb, br):

clone\_population = Population()

clone\_population.individuals = []

for antibody in population\_of\_antibodies.individuals:

clone\_population.individuals = []

alpha = math.exp(-0.1 \* affinity(antibody, antigen))

for c in range(nc):

clone\_population.individuals.append(Antibody(antibody.x + alpha \* random.uniform(-1, 1), antibody.y + alpha \* random.uniform(-1, 1)))

clone\_population.individuals.sort(key=lambda a: affinity(a, antigen), reverse=False)

population\_memory\_cells = Population()

population\_memory\_cells.individuals = clone\_population.individuals[:nd]

for i in range(nd):

if affinity(population\_memory\_cells.individuals[i], antigen) < bb:

population\_memory\_cells.individuals = population\_memory\_cells.individuals[:i]

break

population\_memory\_cells.reduce\_population(br, self.f)

return population\_memory\_cells

def immune\_network\_algorithm\_init(self, a, b, size\_population\_of\_antibodies, size\_population\_of\_antigens):

population\_of\_antibodies = Population()

population\_of\_antibodies.create\_population\_of\_antibodies(size\_population\_of\_antibodies, a, b)

population\_of\_antigens = Population()

population\_of\_antigens.create\_population\_of\_antigens(size\_population\_of\_antigens, a, b)

return population\_of\_antibodies, population\_of\_antigens

def affinity(individual1, individual2):

return (individual1.x - individual2.x)\*\*2 + (individual1.y - individual2.y)\*\*2